Printemps 2015

Projet [IA02]

Chicago Stock Exchange

DUVAL ALEXANDRA
LAVIOLETTE ETIENNE

utc
Université de Technologie
Compiègne

Table des matières

I. Description du jeu :

Le Chicago stock exchange est un jeu de plateau basé sur les principes de la bourse et de la finance. Deux joueurs s'affrontent et se partagent un pion « trader ». Il permet de se placer sur des piles de marchandises afin de se constituer une réserve personnelle. A chaque tour, un joueur déplace le pion trader sur une pile et opère une action sur la marchandise au-dessus de la pile à gauche du trader et pareillement concernant la pile de droite. A chaque tour, un joueur garde une marchandise dans sa réserve et vend la seconde. La marchandise vendue fait baisser d'un point son cours à la bourse. S'enrichir en appauvrissant l'adversaire … Tel est l'objectif de ce jeu.

Matériel à disposition :

- 6 Marchandises différentes (blé, sucre, riz, café, cacao, maïs),
- 4 jetons de chaque marchandise (36 jetons en tout),
- 1 jeton trader,

Déroulement d'une partie :

A chaque tour, un joueur déplace le pion trader dans le seul sens autorisé (à définir arbitrairement) sur une pile de marchandise étant situé 1, 2 ou 3 piles plus loin.
Il effectue une action autorisée sur chaque marchandise du dessus des piles adjacentes à la position à laquelle le trader est désormais placé.
Il en garde une au choix et vend l'autre. Ainsi il étoffe sa réserve personnelle et fait baisser d'un point le cours de la marchandise vendue.

Le jeu s'arrête dès qu'il ne reste plus que deux piles en jeu.

Un décompte des points est effectué pour déterminer le vainqueur. Ainsi les réserves personnelles de chaque joueur sont évaluées (en fonction du dernier cours de la bourse). Chaque joueur cumule donc un score et le vainqueur est celui ayant le plus gros score.

Note : l'explication du jeu ici présent est celui du Chicago stock exchange à deux joueurs

Structures de données :

- Trader : un entier compris entre 1 et9
- Chaque réserve : une liste d'éléments [ble, mais, sucre]
- La bourse : une liste de couples : [[ble, 5], [cacao, 2], [cafe, 5], [mais, 0], [riz, 6], [sucre, 2]]
- Une pile : une liste d'au max 4 éléments : [mais, ble, sucre, riz]
- Les piles : une liste composée de 9 piles (9 listes)

II. Principaux prédicats et gestion Humain vs Humain :

A. Implémentation du menu :

L'implémentation d'un menu en Prolog fut un exercice inhabituel. L'absence de boucle if (ou la possibilité d'utiliser de case) nous oblige à repenser totalement la manière dont on conçoit les menus « habituellement ».
Nous avons découpé cette tâche en différents prédicats ayant chacun un but précis :

- *Boucle lecture du choix :*

Afin d'empêcher l'utilisateur de rentrer des choix différents de 1, 2, 3, 4, nous avons créé ce prédicat pour boucler tant que les deux conditions à la fin du repeat ne sont pas remplies :

```
lecture_choix(Choix):-repeat,
        read(Choix),
        (Choix >0;Choix <4).
```

L'écriture de ce prédicat nous a amené à réfléchir sur l'existence de boucles en Prolog. Même si la récursivité est primordiale en Prolog nous avons jugé utile l'utilisation du prédicat « repeat » qui va boucler sur « read(Choix) » tant que l'une des deux conditions n'est pas remplie. Ainsi le choix fait par l'utilisateur est cohérent et est renvoyé par ce prédicat pour être utilisé ensuite.

- *Lancement du start selon le choix :*

Afin de prendre en compte le choix de l'utilisateur nous avons écrit le prédicat précédent. Il nous fallait désormais en coder un qui lançait le bon « start », c'est-à-dire celui correspondant au choix de l'utilisateur récupéré précédemment. Soit le start Humain vs Humain, soit celui d'Humain vs IA, soit celui d'IA vs IA soit un choix pour quitter.

Voici le prototype de ce prédicat qui ne prend aucun paramètre mais est simplement considéré comme un starter.

```
choix_menu(1):- start2, !.
choix_menu(2):- start3, !.
choix_menu(3):- start4, !.
choix_menu(4):- nl,nl, write('A bientot...').
```

- *Le menu lui-même :*

Le menu en lui-même n'est qu'un prédicat sans argument qui affiche et appelle « lecture_choix « puis « choix_menu »

```
menu:-nl, write('Bievenue dans le Chicago Stock Exchange'),nl,nl,
      write('1. Homme vs Homme'),nl,
      write('2. Homme vs IA'), nl,
      write('3. IA vs IA'), nl,
      write('4. Quitter'),nl,nl,
      lecture_choix(Choix),
      choix_menu(Choix).
```

Critique : Nous aurions pu implémenter un menu plus « user-friendly », notamment esthétiquement. Un module de choix de la difficulté de l'IA n'est pas présent ici. Aussi il n'y a pas de boucle permettant d'enchainer les parties. C'est peut-être le problème principal. Les parties se jouent une à une sans possibilité de les enchainer sans relancer le prédicat « menu ».

B. Générations aléatoires :

Nous avons dû générer aléatoirement :

- La position du trader (même si cela n'était pas nécessaire, ça nous a semblé plus clair)
- Les piles de marchandises selon les contraintes : exactement 6 jetons de chaque marchandise et une génération de 9*4 piles exactement.

- *Génération de la position du trader :*

```
fGenTrader(X) :- random(1,10,X).
```

Une simple génération d'un nombre aléatoire grâce à la fonction random entre 1 et 10. La borne supérieure n'étant pas incluse alors que l'inférieure oui.

- *Génération de la liste des piles de marchandises :*

```
initMarchandises([P1,P2,P3,P4,P5,P6,P7,P8,P9]):-initMarchandisesDisponibles(R),initPile(R,P1,R2)
initPile(R2,P2,R3),
initPile(R3,P3,R4),
initPile(R4,P4,R5),
initPile(R5,P5,R6),
initPile(R6,P6,R7),
initPile(R7,P7,R8),
initPile(R8,P8,R9),
initPile(R9,P9,_).
```

Utilisant le prédicat d'initialisation des piles :

```
initPile(Ressources,[J1,J2,J3,J4],NRessources):-
```

Qui se sert de l'extraction d'une marchandise :

```
extraire([T|_], 0, T):-!.
extraire([_|Q], I, Res):-J is I-1, extraire(Q,J,Res).
```

De la liste codée en dure composée de toutes les marchandises :

```
initMarchandisesDisponibles
```

La génération de la pile de marchandise génère chacune des 9 sous listes en cascade. En effet, la contrainte de « 6 jetons exactement répartis dans 9 piles de 4 marchandises » nous oblige à passer successivement le résultat de l'initialisation d'une pile dans la suivante. Les variables globales n'existant pas nous n'avons pas trouvé d'autre solution.

L'initialisation d'une pile se fait grâce au passage en paramètre de la liste « initmarchandisedispo » (au premier appel), on calcule la longueur de cette liste, on choisit un nombre aléatoire entre 1 et la longueur de la liste. On extrait cet élément. On réitère le processus 4 fois. Quatre éléments que l'on place dans une liste de retour. On n'oublie pas de renvoyer en retour la nouvelle « initmarchandisedispo » amputée de ses 4 éléments… Ainsi on se retrouve au final avec une liste de 32 éléments (à la fin du premier appel) et une première pile constituée. Ainsi de suite 9 fois de suite.

C. Prédicats divers :

Quelques prédicats ont été implémentés de manière générale pour des besoins divers et ne sont pas destinés à une tâche précise. En voici ici une description brève.

- *Concaténation :*

L'implémentation très simple du prédicat de concaténation de deux listes dans une troisième. Il nous sert à plusieurs reprises dans nos boucles récursives (notamment la mise à jour des réserves des joueurs après sélection de la marchandise à garder).

```
concat([], L, L).
concat([T|Q], L, [T|R]):-concat(Q, L, R).
```

- *Premier élément et queue d'une liste :*

Nous avons décidé de rajouter un prédicat nommé « first », celui-ci unifie le second paramètre passé au prédicat avec la tête de la liste. Nous récupérons ainsi la tête de la liste sous forme d'un atome directement utilisable.

```
first([H|_],H).
```

De même nous avons eu besoin, lors de la mise à jour des piles, de ne travailler qu'avec la queue d'une liste, le prédicat « queue » nous permettait d'unifier directement un paramètre de ce prédicat avec la queue d'une liste passée elle aussi en paramètre.

```
queue([_|Q], Q).
```

D. Prédicats d'affichage :

Afin de réaliser les différents affichages pour montrer l'évolution du plateau ou simplement sa composition, nous avons implémenté divers prédicats pour afficher les éléments du jeu.

- *Afficher le Trader :*

Afin d'afficher le trader avec un signe distinctif à l'affichage du plateau nous avons décidé un affichage en colonne des marchandises et symbole graphique pour indiquer sa position, sur la même ligne que la pile sur laquelle il se situe. Voici le prédicat utilisé : un simple affichage si la position actuelle lors de l'affichage des piles correspond à celui du Trader actuellement :

```
afficher_pos(P,P):-write(' <- TRADER').
afficher_pos(_,_).
```

- *Afficher le premier élément d'une liste :*

Pour afficher les piles de marchandises à l'écran nous avions besoin d'un prédicat prenant une liste en paramètre et n'affichant que le premier élément de celle-ci. Très simplement nous avons choisi :

```
afficher_premier([H|[]]):-write(H).
afficher_premier([H|_]):-write(H).
```

- *Afficher les marchandises :*

Pour afficher selon les conventions les piles de marchandises (le premier élément de chaque sous liste de la liste des piles ainsi que le trader) nous avons choisi d'implémenter un prédicat qui agit successivement sur la tête puis la queue, introduisant un formalisme graphique, un compteur et vérifiant si la position actuelle est celle à laquelle est située le Trader. Ainsi on appelle le prédicat « afficher_pos » qui affiche le Trader si la position actuelle concorde avec celle du Trader :

```
afficher_marchs([],_,_).
afficher_marchs([H|T],P,I):-write(I),write(': '),afficher_premier(H),afficher_pos(P,I), !, nl,J is I+1,afficher_marchs(T,P,J).
```

- *Afficher le contenu d'une liste :*

Pour afficher le contenu des réserves des deux joueurs nous avions besoin d'un simple prédicat parcourant une liste jusqu'à son terme et affichant chaque élément la composant :

```
afficher_liste([]).
afficher_liste([H|T]):-write(H), write(' '), afficher_liste(T).
```

- *Afficher le plateau :*

Le prédicat mentionné comme le premier à écrire lors de la réalisation du projet était celui permettant d'afficher le contenu du plateau.

Pour ce faire, nous avons écrit un prédicat qui prend en paramètre une configuration de plateau :

La pile des marchandises, la bourse actuelle, la position du trader ainsi que les deux réserves.

Nous avons introduit des formalismes graphiques mais surtout l'appel successif des prédicats : « afficher_liste » sur les deux réserves. L'affichage de la bourse passée en paramètre puis l'affichage des piles de marchandises grâce à « afficher_marchs ».

```
affiche_plateau(Pile_march, Bourse, Trader, RJ1, RJ2):-
```

E. Prédicat « Coup_possible » :

Avant de réellement jouer un coup à proprement dit, il nous était imposé de créer un prédicat qui vérifiait si un coup était réalisable. C'est le rôle du prédicat « coup possible ».

En voici son prototype :

```
coup_possible(Pile_march, Bourse, Trader, RJ1, RJ2, Joueur, Deplacement, Garde, Jete, NewTrader):-
```

En voici l'algorithme :

Appeler le prédicat qui demande et vérifie la validité du déplacement choisit par le joueur ;

Calculer la nouvelle position du trader en fonction du déplacement ;

Récupérer la position de la pile de gauche du trader ;

Récupérer la pile de gauche entier ;

Faire de même avec la pile de droite ;

Afficher les premiers éléments de chaque pile récupérée ;

Appeler le prédicat pour garder une marchandise ;

Appeler le prédicat pour jeter (vendre) une marchandise ;

Faire les affichages généraux ;

Voici la description brève des prédicats utilisés dans cette évaluation de la validité du coup :

- *Tester la validité d'un coup*

```
testCoup(Deplacement):-repeat,
  write('De combien voulez-vous vous deplacer ? (1,2,3)'),nl,
  read(Deplacement),
  (Deplacement > 0, Deplacement <4).
```

Ce prédicat répète la demande de saisie d'un déplacement tant que celui-ci n'est pas compris entre 1 et 3 (les deux bornes incluses).

- *Récupérer la position de gauche (resp. de droite)*

```
position_G(Newtrader, P_g, Pile_marc):-
position_G(Newtrader, P_g, Pile_marc):-

position_D(Newtrader, P_d, Pile_marc):-
position_D(Newtrader, P_d, Pile_marc):-
```

Ce prédicat récupère respectivement la position à gauche et à droite du trader. Il faut vérifier si nous ne sommes pas arrivé au bout de la pile de marchandise (et ainsi renvoyer « 1 »).

- *Prédicat pour garder une marchandise*

```
garder_Marchandise(Choix1, Choix2, Choix_garder):-repeat,
    write('Quelle marchandise souhaitez vous garder ?'), nl,
    read(Choix_garder),
    (Choix_garder = Choix1;Choix_garder = Choix2).
```

Ce prédicat demande à l'utilisateur la marchandise à garder et boucle tant que la réponse de l'utilisateur n'est pas exactement l'un des deux choix proposé précédemment (l'un des deux premiers jetons des piles adjacentes).

- *Prédicat pour vendre une marchandise*

```
jeter_Marchandise(Choix1, Choix2, Garde, Choix2):- Choix1 = Garde,!.
jeter_Marchandise(Choix1, Choix2, Garde, Choix1):- !.
```

Ce prédicat s'exécute après le précédent et enregistre automatiquement la marchandise vendue. En effet, nous avons voulu éviter les erreurs de saisie et ainsi ne demande que la marchandise à garder après avoir proposé les deux marchandises « choix ».

F. Prédicats de mise à jour :

Afin de jouer un coup il nous fallait mettre à jour les différents éléments. Ici se situe sans doute les prédicats ayant posés le plus de soucis en conception. Comment mettre à jour une bourse de manière cohérente ? De même avec la liste des marchandises…

- *Mise à jour des réserves :*

Nous avons décidé d'implémenter deux prédicats de mise à jour des réserves. Ce choix est contestable mais nous avons décidé de découper le plus possible nos mise à jour pour avoir un prédicat global clair. Il existe donc deux versions du prédicat de mise à jour des marchandises.

Les paramètres passés à ces prédicats sont :

Un joueur, une marchandise (gardée en l'occurrence), une ancienne réserve et la nouvelle réserve de sortie.

Dans le cas où l'on souhaite mettre à jour la réserve du joueur 1. On test si le joueur passé en paramètre est le 1 ou le 2... Dans le cas du joueur 2, rien n'est effectué. La nouvelle réserve est l'ancienne. Dans le cas contraire, on renvoie une liste qui concatène la réserve gardée dans l'ancienne réserve et ainsi renvoie cette nouvelle réserve.

Le même raisonnement est tenu pour la mise à jour de la réserve du joueur2.

```
maj_reserve_joueur1(Joueur, Garde, OldReserve, OldReserve):-
maj_reserve_joueur1(Joueur, Garde, OldReserve, NewReserve):-

maj_reserve_joueur2(Joueur, Garde, OldReserve, OldReserve):-
maj_reserve_joueur2(Joueur, Garde, OldReserve, NewReserve):-
```

- *Mise à jour de la bourse :*

Le point crucial pour passer d'un coup à un autre était de tenir la bourse à jour en fonction de la marchandise vendue. Ainsi nous avons construit un prédicat « maj_Bourse » qui prend en paramètre une marchandise, une bourse et renvoie la nouvelle bourse mise à jour.

Pour ce faire, on récupère tout d'abord le cours de la marchandise passée en paramètre grâce à un « select([Marchandise|X], Bourse, Unifie) ». On garde aussi la bourse sans la marchandise à mettre à jour telle quelle dans la variable « Unifie ». On baisse le cours de la marchandise, on reconstitue la liste constituée du nom de la marchandise et de son cours. Cette nouvelle sous liste est concaténée avec la liste bourse « Unifie ».

```
maj_Bourse(Marchandise, Bourse, L1):-
```

- *Mise à jour de la pile des marchandises :*

Le dernier point essentiel de cet ensemble de mises à jour était de reconstituer la liste des marchandises sans les têtes des piles de gauche et de droite par rapport au trader.

Ainsi un prédicat prend en paramètres d'entrée la pile à mettre à jour, la liste des piles de marchandise, la nouvelle pile de marchandise et le numéro de la pile à mettre à jour dans la liste des marchandises (sa position).

On effectue les actions suivantes :

- Récupérer toutes les piles de la liste sauf celle à mettre à jour, les stocker dans une variable (Unifie) ;
- Récupérer la queue de la liste à mettre à jour ;
- On génère par produit cartésien les combinaisons possibles des piles de la variable Unifie avec la pile mise à jour ;
- On sélectionne seulement celle dont la nouvelle liste est à la bonne place grâce à la position passée au paramètre et au « nth ».

```
maj_Piles(Pile_supp, Piles, L1, Numero):-
```

- *Changement du joueur :*

Simple passage du Joueur 1 au 2 si le joueur 1 était l'ancien joueur et vice-versa.

```
nouveauJoueur(Ancien, Nouveau):- Ancien<2, Nouveau is Ancien +1,!.
nouveauJoueur(Ancien, Nouveau):- Nouveau is Ancien -1,!.
```

- *Nouvelle position du Trader :*

Voici le prédicat qui nous a posé le plus de difficulté en raison des différents cas possibles. Ce prédicat prend une pile de marchandises en paramètre, une ancienne position de trader, un déplacement et renvoie la nouvelle position du Trader.

```
new_Pos_Trader(OldTrader, Deplacement, NewTrader, Pile_march):-
```

Pour ce faire, nous avons déterminé plusieurs cas possibles :

- Soit la somme du déplacement et de l'ancienne position de trader est <= à la longueur de la pile de marchandise, aucun soucis dans ce cas. Le nouveau trader est simplement la somme du déplacement et de l'ancienne position du trader.
- Soit la somme dépasse la longueur de la liste alors il faut vérifier si l'ancienne position du trader dépasse la longueur de la liste ou pas. Alors le nouveau trader est : (Oldtrader + Déplacement) – longueur de la liste,
- Soit l'ancien trader dépasse la longueur de la liste, alors on décrémente l'ancien trader et on fait: (Oldtrader + Déplacement) – longueur de la liste.

Critique : Ce prédicat omettait certains cas spécifiques que nous avons préféré coder en dur (l'apparition de scénarios impossibles bloquant le jeu). Nous n'avons pas trouvé les conditions exactes pour construire le « new_pos_Trader » de façon optimale. Sans doute que notre choix de supprimer les listes vides seulement après cette mise à jour de position de trader nous bloquait et rendait le problème beaucoup plus compliqué.

G. Prédicat pour « jouer_coup » :

Après avoir introduit le prédicat qui détermine si un coup est possible il faut le jouer. Et ainsi utiliser les prédicats de mise à jour ainsi que d'autres prédicats annexes présentés succinctement ci –après.

Prototype :

```
jouer_coup(Pile_march, Bourse, NewTrader, RJ1, RJ2, Joueur, Deplacement,
Garde, Jete, NewTrader5, NewPile_march2, NewBourse, NewReserveJ1, NewReserveJ2):-
```

Algorithme :

Mettre à jour la réserve du joueur1 ;

Mettre à jour la réserve du joueur2 ;

Mettre à jour la bourse ;

Récupérer la position de gauche du trader, la pile de gauche ;

Faire de même à droite ;

Mettre à jour la pile avec les infos de la pile de gauche ;

Faire de même avec les infos de la pile de droite ;

Tester si la pile de gauche du trader est vide ;

Tester si on est au bout de la liste des marchandises et que la première devient vide ;

Vérifie que le trader n'est pas égal à 0 ;

Supprimer les listes vides (s'il y en a) dans la liste des marchandises ;

Vérifier si le trader n'a pas une position plus grande que la longueur de la nouvelle liste des marchandises ;

De nombreuses vérifications intermédiaires ont été mises en place afin de ne pas aboutir à des situations incohérentes. On a fait appel à de nombreux prédicats définit dans la section des prédicats de mise à jour. Les autres sont ceux-ci :

- *Supprime les listes vides :*

```
supp_liste_vide([], []).
supp_liste_vide([[]|Q], N_Pile):-
supp_liste_vide([T|Q], N_Pile):-
```

Les paramètres d'entrée sont : une liste de départ et la liste sans les listes vides de sortie.

Ce prédicat parcourt la liste récursivement jusqu'à son terme. Puis, au retour des appels récursifs, teste si la tête est une liste vide. Si ce n'est pas le cas, on construit la liste de retour avec la tête et ainsi de suite.

- *Vérifie si la pile de gauche du trader est vide :*

```
pile_gauche_vide(Pile, OldTrader, NewTrader):-
pile_gauche_vide(Pile, OldTrader, OldTrader).
```

Décrémente la position du trader si la pile de gauche est vide. Sinon aucune opération n'est effectuée au niveau de la position du trader.

- *Vérifie si la première pile est vide lorsqu'on arrive au bout de la liste des marchandises :*

```
pile_droit_au_bout(Pile, Pile_Droite, OldTrader, NewTrader):-
pile_droit_au_bout(Pile, Pile_Droite, OldTrader, OldTrader).
```

Vérifie si la première pile est vide au cas où le trader se situe au bout de la liste des marchandises. Décrémente la position du trader le cas échéant. Ne fait rien sinon.

- *Vérifie que le trader n'est pas égal à zéro :*

```
trader_zero(Old, New):- Old =:= 0, New is 1, !.
trader_zero(Old, Old).
```

Simple prédicat qui remet la valeur du trader au début de la pile si par mégarde il arrive à 0.

- *Vérifie que le trader ne dépasse pas la longueur de la liste des marchandises :*

```
trader_sup(Old, New, Pile):- length(Pile, Longueur), Old > Longueur, New is Longueur, !.
trader_sup(Old, Old, Pile).
```

Simple prédicat qui repositionne le trader sur la dernière pile si par mégarde il a une position supérieure.

H. Prédicats de fin de jeu :

A la fin du jeu, une batterie de prédicats se met en marche pour réaliser différentes actions :

- Afficher la bourse et les réserves après le dernier coup,
- Transformer la liste des marchandises en une liste des cours leur correspondant
- Faire la somme de ces listes et renvoyer le résultat dans des scores
- Déterminer le gagnant

- *Fin_du _jeu :*

```
fin_jeu(Bourse, RJ1, RJ2):-
```

Ce prédicat, en dehors des affichages purement graphiques, effectue les actions suivantes :

- Affiche la dernière bourse,

- Transforme les listes de marchandises en listes de cours correspondantes
- Fais la somme de ces listes pour déterminer le score,
- Appelle le prédicat « winner » pour décider qui a gagner

- *Transformation des réserves:*

```
liste_Cours([],_,[]):-!.
liste_Cours([T|Q], Bourse, Liste_sortie):-
```

Ce prédicat prend en paramètre, une réserve, une bourse et renvoie en sortie une liste des cours. Pour ce faire on parcourt la liste jusqu'au bout de manière récursive, puis pour chaque marchandise trouvée (à chaque fin d'appel récursif), on sélectionne dans la bourse la valeur du cours correspondante à la marchandise. Valeur que l'on ajoute (concaténation) dans la liste de sortie.

- *Somme des éléments d'une liste :*

Pour calculer le score d'un joueur, nous additionnons les valeurs de la liste des cours que nous avons construit grâce au prédicat « liste_cours ».

```
somme([],0).
somme([X|R],N) :- somme(R,N1), N is N1+X.
```

Le parcours récursif de la liste passée en paramètre jusqu'au bout nous permet ensuite de construire par addition la valeur final de la somme.

- *Déterminer le gagnant :*

```
winner(ScoreJ1, ScoreJ2):-
winner(ScoreJ1, ScoreJ2):-
winner(ScoreJ1, ScoreJ2):-
```

Afin de conclure correctement le jeu, il nous fallait implémenter précisément un prédicat final, appelé en dernier, qui se chargeait d'effectuer les affichages et les comparaisons des scores passés en paramètres.

Trois scénarios (implémentés grâce au cut « ! » symbolisant le « if-elsif-else ») ont été dégagés : soit la victoire du joueur1, soit celle du joueur2, soit l'égalité. Les affichages en conséquent sont implémentés dans chaque scénario.

I. Moteur et enchainement Humain vs Humain :

Afin de faire enchainer les coups entre deux humains nous avons implémenté le prédicat « suivant » dont voici le prototype :

```
suivant(Pile_march, Bourse, Trader, RJ1, RJ2, Joueur_act):-
```

Son fonctionnement est le suivant :

Si le nombre de piles < 3 alors déclencher les prédicats de fin du jeu ;

Sinon

Changer le joueur ;

Demander le coup suivant et tester s'il est possible « coup_possible » ;

Jouer le coup ;

Appeler ce prédicat « suivant » sur le nouveau plateau

Nous avons aussi ajouté un prédicat start2 pour générer un plateau et l'envoyer dans le prédicat suivant pour faire jouer les coups aux deux humains. Le voici :

```
start2:-plateau_depart(Pile_march, Bourse, Trader, RJ1, RJ2),
        suivant(Pile_march, Bourse, Trader, RJ1, RJ2, 0).
```

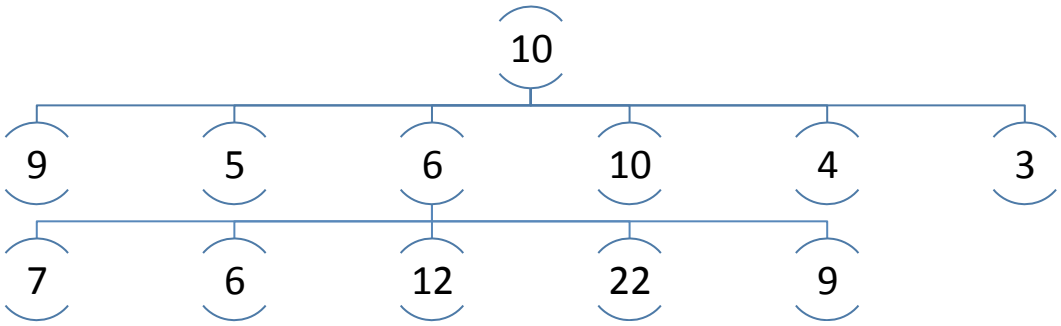
Nous utilisons notre astuce d'appeler le prédicat avec une valeur de joueur à 0 pour marquer le coup et bien préciser que c'est le premier appel. Ainsi avec la fonction « nouveauJoueur » nous allons faire passer cette valeur à 1. Et ainsi pouvoir enchainer les coups entre les deux joueurs jusqu'à la fin du jeu.

III. Intelligence Artificielle :

A. Description de l'IA utilisée

Dans ce projet nous avons décidé d'implémenter l'algorithme « MinMax » présenté par notre responsable Kevin Carpentier en séance de TP. Ses explications nous ont permis d'adapter ce fameux algorithme à notre Chicago Stock Exchange.

En voici la représentation que nous en avons faite :



Comme précisé sur le schéma ci-dessus, nous nous sommes limités à la simulation des coups à une profondeur 2. Ce fut un choix arbitraire, pour des facilités de programmation mais aussi d'un point de vue logique. Ainsi nous avons évité les problèmes de surcharge mémoire et de lenteur de l'algorithme. Nous sommes partis du postulat que la profondeur 2 nous permettait un compromis intéressant entre performance et utilisation de ressources.

Deux cas sont présents dans notre implémentation :

- Lorsque l'IA est appelée par le joueur 1
- Lorsque l'IA est appelée par le joueur 2

Ces deux cas ont nécessité une implémentation différente. Comme représenté sur l'arbre ci-dessus, lorsque le joueur 1 joue l'IA simule les coups de profondeur 1 et les 36 autres résultats de chaque possibilité à la profondeur 1. L'IA s'occupe de calculer le score du joueur 1 à la profondeur 2 (avec une simple mise à jour de la bourse et un calcul du score par rapport à la réserve du joueur 1 et à cette nouvelle bourse). Tous ces scores sont ensuite

remontés à la profondeur 1 où l'IA s'occupe de sélectionner le max. En effet on estime que le joueur 2 jouera le coup optimal quand ce sera son tour. Il va donc chercher à minimiser le score du joueur 1. Il faut donc prendre une branche du graph où l'on va maximiser le score du joueur 1. Evidemment il est possible que le joueur 2 (surtout si c'est un humain) joue un coup non optimal, alors il décuplera les chances de victoire du joueur 1.

Lorsque le joueur 2 joue, nous avons décidé de ne pas manipuler que les scores du joueur 1. L'implémentation mise en place est donc défensive du point de vue du joueur 2. Celui-ci va s'occuper de minimiser le score du joueur 1. Il faut donc inverser les minimums et maximum calculés dans la première configuration possible. En effet à la profondeur 2, lorsque c'est au joueur 1 de jouer nous avons calculé les scores des 36 configurations possibles. Nous en avons extrait le maximum pour chaque « six-uplet ». Ces scores sont remontés à la profondeur 1. A cette profondeur 1 nous avons sélectionné le minimum des maximums calculés précédemment. En effet il est d'intérêt pour le joueur 2 de prendre la branche qui amène le joueur 1 au plus faible maximum lors du coup suivant. Encore une fois nous estimons que le joueur 1 dans ce cas précis va jouer le coup optimal.

B. Prédicats d'intelligence artificielle

- ***Coups possible :***

```
coups_possible(Pile_march, Bourse, Trader, RJ1, RJ2, Joueur, [[1, March_G_1, March_J_1], [1, March_G_2, March_J_2],
                                                              [2, March_G_3, March_J_3], [2, March_G_4, March_J_4],
                                                              [3, March_G_5, March_J_5], [3, March_G_6, March_J_6]]):-
```

Ce prédicat prend comme paramètres d'entrée :

- La pile de marchandises,
- La bourse actuelle,
- Le Trader,
- La réserve du joueur 1,
- La réserve du joueur deux
- Le joueur actuel

Et renvoie une liste constituée de 6 sous-listes. Cette liste sera constituée comme suit : [[déplacement suivant, marchandise gardée, marchandise vendue], …].

Grâce à ce prédicat nous allons opérer comme suit :

Algorithme :

Pour chaque déplacement possible appartenant à {1, 2,3} faire
Récupérer la nouvelle position du trader ;
Récupérer la position de gauche du trader actualisé ;
Récupérer la position de droite du trader actualisé ;
Récupérer la pile de droite correspondant à la position droite ;
Récupérer la pile de gauche correspondant à la position gauche ;
Récupérer les deux marchandises en tête de ces piles
Les unifier avec les éléments des sous listes correspondants pour le retour
FinPour

Note : On a codé en « dur » les déplacements possibles : ainsi on s'occupe de traiter les deux cas possibles concernant le déplacement de 1. Puis les deux cas pour le déplacement égal à deux et pareil pour le dernier cas possible de déplacement.
En notant que les deux scénarios possibles pour chaque déplacement ne sont différents que par la marchandise gardée dans un cas, jetée dans l'autre, on a évité d'avoir six blocs répétitifs. Ainsi, ici seulement trois grands blocs se dégagent.

Critique : On a codé directement le contenu de la liste de retour : une grande liste composée de 6 sous listes. Ce prédicat n'est pas réutilisable. Au contraire, il est totalement écrit sur mesure pour le Chicago Stock Exchange. Une forme générique aurait pu être utilisée. Aussi le calcul de la position, la récupération de la pile puis la récupération du premier élément de cette pile auraient pu être combinées dans un « super-prédicat ».

Complexité : Aucun appel récursif, simplement l'exécution séquentielle des prédicats intermédiaires.

- *Meilleur coup possible :*

  Afin d'écrire ce prédicat nous avons délégué certaines tâches à des prédicats intermédiaires écrits par nos soins selon les besoins.

- *Minimum/Maximum d'une liste :*

Afin d'appliquer l'algorithme « MinMax » nous avons besoin de sélectionner le plus grand (resp. Petit) élément d'une liste de 6 nombres (dans notre cas précis) passée en paramètre.

```
maximum_elements(ScoreJ1, ScoreJ2, ScoreJ3, ScoreJ4, ScoreJ5, ScoreJ6, Max):-
minimum_elements(ScoreJ1, ScoreJ2, ScoreJ3, ScoreJ4, ScoreJ5, ScoreJ6, Min):-
```

Pour ce faire, nous comparons séquentiellement deux à deux le score du premier joueur avec le score du second. Les prédicats de GNU Prolog Max(Nombre1, Nombre2) et Min(Nombre1, Nombre2) nous permettent de faire cette comparaison. Nous comparons ce maximum/minimum avec le score du joueur 3. Nous stockons ce nouveau max dans une variable intermédiaire et ainsi de suite.

La dernière comparaison est le max/min.

Critique : Une nouvelle fois, un prédicat sur mesure et cinq utilisations de Max/Min ainsi que de nombreuses variables intermédiaires. Une solution qui fonctionne mais ne semble pas être la plus optimisée.

- *Sélection du « bon » coup :*

Afin d'appliquer l'algorithme « MinMax » nous avons eu besoin de renvoyer en paramètre ce coup optimum. C'est-à-dire le déplacement, la marchandise gardée, la marchandise vendue. Pour ce faire, nous avons écrit un prédicat qui unifie les paramètres d'entrée d'un coup avec les paramètres de sortie si et seulement si le score de ce coup est le score maximum. Ainsi c'est une sorte de sélecteur de coup.

```
test_bon_coup(Score, Max, D, Garde, Jete, NewTrader, D, Garde, Jete, NewTrader):- Score= Max, !.
test_bon_coup(Score, Max, D, Garde1, Jete1, NewTrader1, Deplacement, Garde, Jete, NewTrader).
```

Critique : Ce prédicat a besoin d'être appelé sur les six coups candidats à être le meilleur coup. Cinq fois sur six le prédicat ne fait rien. Parfois si deux coups sont équivalents seul le premier sera considéré comme meilleur coup possible. En effet les valeurs de sorties de meilleur coup possible seront unifiées avec les premières valeurs de « test_bon_coup ». Ce fut un choix de notre part. On a placé un « cut » d'optimisation afin d'empêcher de créer et d'explorer les branches inutiles.

- *Sélection du « bon » coup :*

Afin de simuler les coups à la profondeur 2 ainsi que le calcul des scores de chaque simulation pour le joueur 1 nous avions besoin de ce prédicat primordial.

```
joueur_IA(1, NewBourse, NewReserveJ1, NewReserveJ2, Coups_possibleProf, ScoreJ1, ScoreJ2, ScoreJ3, ScoreJ4, ScoreJ5, ScoreJ6):-
joueur_IA(2, NewBourse, NewReserveJ1, NewReserveJ2, Coups_possibleProf, ScoreJ1, ScoreJ2, ScoreJ3, ScoreJ4, ScoreJ5, ScoreJ6):-
```

Ce prédicat est construit sur la forme d'un « ou » exclusif. Soit c'est au joueur 1 de jouer le coup suivant. A la profondeur 2 ce sera ainsi au joueur 2 de jouer et nous avons besoin d'effectuer les actions suivantes :

Algorithme (joueur1) :

Pour chaque coup possible de la liste des coups entrée en paramètre

Récupérer les différents éléments : déplacement, marchandise gardée, marchandise vendue ;

Mettre à jour la bourse selon la marchandise vendue (par le joueur 2 en l'occurrence)

Calculer les 6 scores possible en fonction de la réserve du joueur 1 et de la nouvelle bourse actualisée ;

Fin Pour

Renvoyer les 6 scores ;

La deuxième version est utilisée lorsque c'est au joueur 2 de jouer. Ainsi à la profondeur 2 ce sera au joueur 1 de jouer. Comme nous nous intéressons seulement au score du joueur 1, les manipulations à effectuer diffèrent de la version ci-dessus ;

Algorithme (joueur2) :

Pour chaque coup possible de la liste des coups entrée en paramètre

Récupérer les différents éléments : déplacement, marchandise gardée, marchandise vendue.
Mettre à jour la bourse selon la marchandise jetée
Mettre à jour la réserve du joueur avec la marchandise gardée,
Calculer le score correspondant

FinPour

Renvoyer les 6 scores

```
meilleur_coup_possible(Pile_march, Bourse, Trader, ReserveJ1, ReserveJ2, 1 , Deplacement, Garde, Jete, NewTrader):-
meilleur_coup_possible(Pile_march, Bourse, Trader, ReserveJ1, ReserveJ2, 2 , Deplacement, Garde, Jete, NewTrader):-
```

Nous distinguons les paramètres d'entrée :
La pile de marchandise, la Bourse, le Trader, la Réserve du joueur1, la Réserve du joueur2 ainsi que le joueur jouant ce coup (1 pour le « meilleur_coup possible » lorsque c'est au tour du joueur 1 de jouer, 2 quand c'est au tour du joueur 2).

En sortie nous aurons : le déplacement correspondant à ce meilleur coup ainsi que la marchandise gardée, celle qui est vendue (« Jete ») et la nouvelle position du Trader.

Nous avons fait le choix de renvoyer la marchandise gardée et celle vendue pour éviter de calculer à nouveau la marchandise vendue plus tard dans la succession des coups alors que l'on a l'information disponible ici.
Nous avons aussi décidé de renvoyer la nouvelle position du Trader directement pour simplifier notre moteur d'enchainement des coups.

Deux versions sont ici aussi distinguées :

- Lorsque c'est au tour du joueur 1 :

Algorithme (tour joueur1) :

Générer la liste des coups possibles disponibles selon le plateau passé en paramètre ;

Pour chacun des 6 coups de cette liste faire
Récupérer les informations de déplacement, marchandise gardée, marchandise vendue ;
Calculer la nouvelle position du Trader en fonction de celle initiale (passée en troisième paramètre) et le déplacement récupéré ci avant ;
Jouer ce coup ;
Se servir des informations du nouveau plateau pour générer la liste des coups disponibles (une nouvelle fois 6 coups disponibles) ;
Appel du prédicat « joueur_ia » version 1 pour récupérer les 6 scores possibles ;

Appel de « Minimum_éléments » pour récupérer les minimums de ces 6 scores et les remonter dans le graph ;

FinPour

Appel de « Maximum_éléments » sur les minimums et récupérer ce maximum ;

Appel de « test_bon_coup » sur les 6 coups possibles de profondeur 1 et ainsi unifier les paramètres de sortie en fonction du coup correspondant au maximum des minimums.

Critique : Cet algorithme n'est pas extensible et difficilement modifiable, nous avons décidé de coder directement une recherche à la profondeur 2. Nous avons décidé de faire appel à « jouer_coup » pour simuler chaque coup et anticiper les prochains. Cependant ce prédicat n'était destiné qu'à la réalisation d'un coup. Nous l'avons ici détourné de sa fonction première.

Lorsque c'est au tour du joueur 2 :

Algorithme (tour joueur2) :

Générer la liste des coups possibles disponibles selon le plateau passé en paramètre ;

Pour chacun des 6 coups de cette liste faire

Récupérer les informations de déplacement, marchandise gardée, marchandise vendue ;
Calculer la nouvelle position du Trader en fonction de celle initiale (passée en troisième paramètre) et le déplacement récupéré ci avant ;
Jouer ce coup ;
Se servir des informations du nouveau plateau pour générer la liste des coups disponibles (une nouvelle fois 6 coups disponibles) ;
Appel du prédicat « joueur_ia » version 2 pour récupérer les 6 scores possibles ;
Appel de « Maximum_éléments » pour récupérer les maximums de ces 6 scores et les remonter dans le graph ;

FinPour

Appel de « Minimum_éléments » sur les maximums et récupérer ce minimum ;

Appel de « test_bon_coup » sur les 6 coups possibles de profondeur 1 et ainsi unifier les paramètres de sortie en fonction du coup correspondant au maximum des minimums.

Ainsi avec ce prédicat à deux versions possibles nous avons pu implémenter le comportement offensif de l'IA (joueur 1) et le comportement défensif de l'IA (joueur 2). Il nous reste à implémenter le moteur IA vs IA en articulant les coups du joueur 1 et du joueur 2.

- *Moteur de l'IA :*

Une fois tous les prédicats implémentés, il nous fallait créer un moteur d'enchaînement des coups. Certes, celui-ci reprend la structure de celui utilisé pour le moteur Humain vs Humain et de nombreux prédicats définis et utilisés précédemment mais c'est une part importante et la finalisation de notre travail.

```
suivant_IA(Pile_march, Bourse, Trader, RJ1, RJ2, Joueur_act):-
```

Ce prédicat prend un simple plateau en paramètre. A noter qu'une nouvelle fois, le « Joueur_Act » passé en paramètre sera égal à 0 au premier appel. En effet nous décidons de passer le joueur précédent en paramètre et de calculer la valeur du joueur actuel dans le prédicat lui-même. Le choix de 0 est ici pour signifier que c'est le premier appel du prédicat. Nous avons hésité à passer 2 en paramètre et ainsi se retrouver avec le joueur 1 en action après l'appel du prédicat « nouveauJoueur » cependant cela aurait pu porter à confusion.

Algorithme :

Si le nombre de piles en jeu est < 3

Appeler le prédicat de fin du jeu ;

Sinon

Afficher le plateau ;

Calculer le nouveau joueur ;

Appeler le prédicat « meilleur_coup_possible » pour le joueur 1

Calculer la nouvelle position du Trader ;

Jouer ce coup ;

Afficher le plateau de « l'entre-deux » coup ;

Appeler le prédicat « next_ia_ia » ;

Afin de pouvoir enchainer les coups entre les deux IA (l'offensive et la défensive) nous avons eu besoin d'implémenter un prédicat « next_ia_ia ». Celui-ci va calculer si après le coup du joueur 1 (IA offensive), la partie n'est pas arrivée à son terme. Si c'est le cas, appel de « suivant_IA » directement.

Si ça n'est pas le cas, il faut faire jouer l'IA défensive et ensuite appeler le prédicat « suivant_IA ».

Voici son prototype :

```
next_test_ia_ia(NewPile_march, NewBourse, NewTrader3, NewReserveJ1, NewReserveJ2, 1):-
```

Cet algorithme effectue les actions suivantes :

Algorithme :

Si la longueur de la liste des marchandises est < 3 alors

Appeler « suivant_IA » avec la bourse, le trader, la pile de marchandise, les réserves et le joueur issues du coup jouer par le joueur 1 ;

Sinon

Appeler « meilleur_coup_possible » pour le joueur 2 ;

Calculer la nouvelle position du Trader ;

Jouer ce coup ;

Appeler « suivant_IA » avec la bourse, le trader, la pile de marchandise, les réserves et le joueur issues du coup jouer par le joueur 2 ;

Critique : Ce prédicat qui correspond à un « if » n'a pas été implémenté tout d'abord mais le problème est apparu lorsque que nous nous sommes rendu compte que nous jouions les coups deux par deux. Nous aurions pu appeler le prédicat suivant avec chacun des joueurs à tour de rôle. De plus nous calculons deux fois la longueur de la pile de marchandise pour ensuite déterminer si nous sommes à la fin du jeu ou non. Ce qui semble lourd.

- *Lancement du moteur de l'IA :*

Le prédicat « start4 » lance le moteur de l'IA si l'utilisateur fait le choix « 4 » au niveau du menu.

```
start4:-plateau_depart(Pile_march, Bourse, Trader, RJ1, RJ2),
        suivant_IA(Pile_march, Bourse, Trader, RJ1, RJ2, 0).
```

Ce prédicat ne prend aucun paramètre.

Le corps de ce prédicat est la génération du plateau de départ (prédicat défini précédemment) et lance le « suivant » sur la « pile_march » générée, la bourse de départ, le trader de départ et les réserves vides des joueurs 1 et 2. Nous précisons ici joueur = 0 comme décrit ci-dessus.

- *IA vs Humain :*

L'IA développée est aussi utilisée à travers le mode de jeu « IA vs Homme ». Ainsi nous avons pu réutiliser le prédicat « meilleur_coup _possible » ainsi que les sous prédicats qui en découlent. Ici le problème a été d'implémenter le fait qu'un coup est le croisement entre celui joué par un joueur humain (considéré comme le joueur1) et l'IA (considérée comme le joueur 2).

```
suivant_Homme_IA(Pile_march, Bourse, Trader, RJ1, RJ2, Joueur_act):-
```

Ainsi voici l'algorithme correspondant :

Algorithme :

Afficher le plateau ;

Déterminer le nouveau joueur (joueur 1) ;

Appel du prédicat « coup_possible » ;

Appel du prédicat pour jouer le coup demandé par l'utilisation ;

Afficher le plateau ;

Appel du prédicat « next_ia_homme » ;

Comme pour le moteur IA vs IA nous avons dû complexifier notre prédicat « suivant_Homme_IA » puisque deux coups sont joués par tour (un du joueur Humain, un autre de l'IA). Il fallait donc les articuler logiquement afin que cela soit fluide. Nous avons implémenté un « next_ia_homme », sur le même modèle que celui implémenté pour le moteur IA vs IA :

```
next_test_ia_homme(NewPile_march, NewBourse, NewTrader3, NewReserveJ1, NewReserveJ2, 1):-
```

En voici son algorithme :

Algorithme :

Si la longueur de la liste des marchandises est < 3 alors

Appeler « suivant_Homme_IA » avec la bourse, le trader, la pile de marchandise, les réserves et le joueur issues du coup jouer par le joueur 1 (Humain) ;

Sinon

Appeler « meilleur_coup_possible » pour le joueur 2 ;

Calculer la nouvelle position du Trader ;

Jouer ce coup ;

Appeler « suivant_Homme_IA » avec la bourse, le trader, la pile de marchandise, les réserves et le joueur issues du coup jouer par le joueur 2 ;

Critique : On note qu'une partie de cet algorithme est similaire à celui nommé « next_ia_ia ». Nous aurions pu factoriser et ainsi simplement modifier l'appel du suivant pour avoir un code plus léger et moins redondant. Aussi les mêmes critiquent que celles faites pour le prédicat « next_ia_i »a sont possibles. A noter que nous n'avons pas eu besoin d'implémenter ce prédicat avant puisque le moteur Homme vs Homme appelle le prédicat

« suivant_homme » avec les joueurs chacun leur tour et ainsi vérifie entre chaque coup de chaque joueur si nous ne sommes pas arrivés à la fin du jeu.

Afin de lancer cet enchainement de coups entre humain et IA, nous avons créé un prédicat appelé à chaque fois que le choix 3 est effectué par le joueur. En voici son prototype :

```
start3:-plateau_depart(Pile_march, Bourse, Trader, RJ1, RJ2),
        suivant_Homme_IA(Pile_march, Bourse, Trader, RJ1, RJ2, 0).
```

Ce prédicat ne prend aucun paramètre, génère le plateau de départ appelle le prédicat suivant correspondant à l'enchainement des coups « Humain vs IA ».

A noter que pour enchainer les coups nous utilisons le prédicat « next » spécialement conçu pour enchainer les coups entre l'Homme (Joueur1) et le joueur 2 qui fera appel à l'IA (défensive).

Critique : Les mêmes critiquent que celles faites pour le prédicat « next_ia_ia » sont possibles. A noter que nous n'avons pas eu besoin d'implémenter ce prédicat avant puisque le moteur Homme vs Homme appelle le prédicat « suivant_homme » avec les joueurs chacun leur tour et ainsi vérifie entre chaque coup de chaque joueur si nous ne sommes pas arrivés à la fin du jeu.

IV. Limites et problèmes rencontrés :

- Le prédicat permettant de déterminer la nouvelle position du Trader fut le plus difficile à implémenter (différentes combinaisons possibles …) et c'est sans nul doute celui qui est central dans notre développement. En effet, c'est le « levier » de transition entre deux coups. Une erreur sur ce coup empêche le bon déroulement de la partie. Cependant son implémentation et la réflexion qui en découle furent très formatrices et permirent de se rendre compte de l'importance de ces « checkpoints » primordiaux.

- La gestion graphique en mode console d'un jeu complet comme celui-ci reste toujours problématique. Comment trouver le juste milieu pour créer un jeu interactif et clair sans s'y perdre.

- La prise en main de Prolog et les contraintes intrinsèques à ce langage ne nous ont pas facilité la tâche mais nous ont permis de développer une manière de réfléchir enrichissante et intéressante. En effet, l'impossible utilisation de variables globales, de variables modifiables fut une problématique à nos débuts dans ce projet,

- Prolog n'étant pas un langage compilé, nous avons dû nous-mêmes rechercher nos erreurs. La fonction trace fut très utile bien que très rigide. Les différentes boucles récursives s'enchainant ou se cumulant rendent l'évaluation et la compréhension du déroulement selon le moteur Prolog difficile. Cependant, encore une fois, c'est un travail que nous n'avons jamais mené avant et cette nouvelle pratique développée lors de ce projet fut formatrice. Sur un programme aussi compliqué il est difficile de « simuler » toutes les possibilités et ainsi être d'être en présence de prédicats robustes (aucune boucle infinie par exemple sur la partie IA vs IA). La prépondérance du modèle théorique était un point nouveau pour nous dans le cadre d'un projet.

- Prolog nous oblige à nous servir de la récursivité de manière presque systématique. Cela nous a permis de consolider nos acquis et nous former à cette manière de raisonner. Tout comme l'utilisation et la manipulation de nombreuses listes plus ou moins complexes,

- Le test de chaque prédicat de manière individuelle a été nécessaire et ne fut pas toujours aisé,

- Quelques paramètres sont codés en dur et ne sont pas extensibles (certaines évolutions de la position du Trader, la constitution de la liste des « coups_possibles » nécessaire au fonctionnement de l'IA), ce qui est à éviter,

- L'utilisation (à bon escient) des « cut » (!) fut un point important et difficile à prendre en main, nous n'excluons pas la possibilité que certains soient inutiles dans nos prédicats.

V. Améliorations :

Ajouter plus de joueurs,

- Intégrer une difficulté d'IA qui pourrait être facilement implémentable : en fonction de la profondeur des coups évalués par l'IA en question. Ainsi l'IA de niveau 1 se contente d'évaluer son score simplement en évaluant les deux coups suivants. L'IA de niveau 2 évaluerait ainsi les combinaisons possibles jusqu'à 3 coups suivant l'actuel et ainsi de suite,

- Une gestion des accents pour un jeu esthétiquement plus attrayant,

- Un « retry » complet du coup si le joueur ne rentre pas un choix de marchandise valide. Ainsi cela permettrait un retour en arrière de la position du Trader et donnerait au joueur une seconde chance. Cela changerait évidemment le jeu et serait à spécifier dès l'énoncé des règles,

- Une meilleure représentation graphique,

- Condenser le code (regrouper les prédicats de mise à jour des réserves par exemple dans un « super-prédicat » parent et ainsi « découper notre code » et éviter les redondances visibles). Créer donc des fonctions générales (jouer un coup par exemple) plus simples et limiter le nombre de lignes de code par fonction),

- Générer une documentation explicite,

- Ajouter un module de choix à l'utilisateur pour savoir qui commence à jouer,

- Permettre d'enchainer les parties sans avoir à relancer le prédicat « menu »,

- Ajouter un prédicat général de vérification des choix de l'utilisateur qui prendrait les bornes en paramètres. Ainsi cela éviterait une duplication inutile,

- Noms de variables/prédicats plus explicites,

Conclusion :

Ce projet fut une très bonne expérience de programmation. Malgré de nombreuses difficultés à comprendre les rouages du langage Prolog à nos débuts, le développement du projet fut un plaisir. Passées les difficultés techniques, Prolog est un langage instinctif et les contraintes évaluées comme telles à notre début ressortent comme des facilités dans le développement. Ce fut aussi la première intelligence artificielle à coder. Les recherches autour de ce domaine ont permis de comprendre les différents systèmes de codages de l'IA dans le cadre d'un jeu relativement simple.